# Compiler Optimizations on an IR

The goal of this project is to compile a small language into a Control-Flow Graph intermediate representation with instructions in Static Single Assignment (SSA) form and implement copy propagation and global common subexpression elimination on it. A basic code skeleton can be found at

https://github.com/smarr/acc-ir-project

This project is optional and the second of two possible options. If you chose not to do a project, the best possible mark will be a 2 (good/"gut" in German). If you chose to do a project (at most one will be considered), submit it, successfully defend it (at a level of good or better), and you achieve at least 40 points in the exam, you overall mark will be the exam mark improved by 1 grade.

The project has to be submitted on Moodle and presented. Up-to-date details will be made available here:

https://ssw.jku.at/Teaching/Lectures/ACC/

Please notify me of issues with the project description, ask questions, or request clarifications via email (stefan.marr@jku.at).

All projects are to be done individually. Teamwork is not permitted.

**Submission Deadline on Moodle:** June 22nd, 2026, 23:59 Linz time.

**Project Presentation Date:** June 23rd, 2026

## 1. The Small Programming Language: Mini Language 2026

Designing programming language is a great hobby for anyone who likes to fully understand how things work. Unfortunately, languages get complex very quickly.

We use the language specified by the following grammar for this project:

```
MiniLang = {VarDecl} MainDecl.
VarDecl = "var" ident ":" Type ";".
Type = ident {"[" number "]"}.
MainDecl = "fn" "main" "(" ")" "{" {VarDecl} StatSeq "}".
StatSeq = Statement {Statement}.
Statement =
      Designator "=" Expression ";"
    | "if" "(" Condition ")" "{" StatSeq "}" {"elseif" "("
Condition ")" "{" StatSeq "}" } ["else" "{" StatSeq "}"]
    | "while" "(" Condition ")" "{" StatSeq "}"
    | "read" Designator ";"
    | "write" Expression ";"
.
Condition = Expression Relop Expression.
Expression = [Addop] Term {Addop Term}.
Term = Factor {Mulop Factor}.
Factor = Designator | number | "(" Expression ")".
Designator = ident { "[" Expression "]" }.
Relop = "==" | "!=" | "<" | ">" | ">=" | "<=".
Addop = "+" | "-".
Mulop = "*" | "/" | "%".
```

The language is designed to be easy to parse and has only integer values. Integers are 32bit values.

The full grammar based on Coco is available here:

https://github.com/smarr/acc-ir-project/blob/main/ir/MiniLang.ATG

Built-in operations

| | |
|---|---|
| `read Designator` | reads an integer from standard in |
| `write Expression` | writes an integer to standard out |

Here is a sample program in our language:
also available here:
https://github.com/smarr/acc-ir-project/blob/main/tests/test.ml

```
var a: int[10][10];
var i: int;
var j: int;

fn main() {
    // read some data
    i = 0;
    while (i < 10) {
        j = 0;
        while (j < 10) {
            read a[i][j];
            j = j + 1;
        }
        i = i + 1;
    }

    // add 1 to every value
    i = 0;
    while (i < 10) {
        j = 0;
        while (j < 10) {
            a[i][j] = a[i][j] + 1;
            j = j + 1;
        }
        i = i + 1;
    }
}
```

## 2. Recommended Approach to the Project

The focus of the project should be on compiling the program to the Control-Flow Graph intermediate representation with instructions in SSA form and then optimizing it. Thus, being able to execute the resulting program is not a goal.

The grammar is kept deliberately simple without functions or function calls, and it allow us to use a basic recursive descent parser. Parser generators such as Coco or ANTLR can be used:

https://ssw.jku.at/Research/Projects/Coco/

I would recommend the following steps:

1. Implement a parser based on the provided grammar, manually, or with a parser generator.
2. Implement a basic symbol table to capture the necessary information about globals and locals and their types.
3. Add code generation to target a Control-Flow Graph with instruction triples in SSA form.
4. Implement copy propagation and global common subexpression elimination.
5. (optionally) Implement a graph-coloring register allocator.

Below, I'll provide more guidance on these steps.

## 3. Implementing a Parser

The provided grammar can be used to generate a parser with Coco. The repository with example code also contains a `MiniLang.java` that accepts the included `tests/*.ml` file. `MiniLang.ATG` defines the basics required for parsing but does not include any further functionality.

I'd suggest that you use the programming language you are most familiar with for this project and use Coco to generate the basic parser as an initial template to work with. C# and Java would be the classic candidates for which Coco can generate code.

If you chose to use Coco or ANTLR, you could implement most of the functionality as part of the attributed grammar, but since it likely will require debugging, I'd recommend using an approach that makes debugging easy for you.

## 4. Implementing a Basic Symbol Table

For the symbol table, I'd suggest that you use the equivalent of a Java LinkedHashMap to represent the information within a scope and use `Obj` and `Struct` nodes to represent the definitions/names and the corresponding type

information.

The implementation should at least do the following:

- Build up a symbol table with `Obj` and `Struct` nodes
- Initialize the universe with the nodes for the int type.

- The parser should check the context conditions for type correctness Thus, it should not allow us for example to define multiple variables with the same name and should ensure that the program is type correct, i.e., read/write statements and assignments are done with integer values.

## 5. Intermediate Representation

Extend your compiler so that it builds a control flow graph (CFG) for the statements of a Mini program. The basic blocks should contain instructions in form of triples in SSA form. Triple means, every instruction has an operation with 2 operands, which can reference a variable, a constant, or some other instruction (see below). Some instructions have operands that are null (denoted by "-").

| | | | |
|---|---|---|---|
| 0 | neg | x neg - | negate |
| 1 | plus | x plus y | plus |
| 2 | minus | x minus y | minus |
| 3 | times | x times y | times |
| 4 | div | x div y | divide |
| 5 | rem | x rem y | remainder |
| 6 | cmp | x cmp y | compare |
| 7 | phi | x phi opds | phi instruction |
| 8 | ld | x load y | load |
| 9 | lr | - lr y | load register y (only used after removal of phis) |
| 10 | lc | - lc y | load constant y (only used after removal of phis) |
| 11 | st | x store y | store y to the memory address denoted by x |
| 12 | ass | x ass y | assign y to the variable x |
| 13 | read | - read - | read integer value |
| 14 | write | - write y | write integer value y |
| 15 | ret | - ret - | return (used at the end of the program) |
| 16 | br | - br y | branch to block y |
| 17 | blt | x blt y | branch to block y if x references "a cmp b" and a < b |
| 18 | beq | x beq y | branch to block y if x references "a cmp b" and a == b |
| 19 | bgt | x bgt y | branch to block y if x references "a cmp b" and a > b |
| 20 | bge | x bge y | branch to block y if x references "a cmp b" and a >= b |
| 21 | bne | x bne y | branch to block y if x references "a cmp b" and a != b |
| 22 | ble | x ble y | branch to block y if x references "a cmp b" and a <= b |

While generating the CFG, also build its dominator tree, which can then be used for optimizations.

You can assume that all variables reside in a single stack frame to which the special register FP (frame pointer) points. Arrays also reside in this stack frame. Their elements are accessed as ld x, y, where x denotes a base register and y denotes either an offset or an index register.

Implement also methods that print the CFG and its instructions so that you can check their correctness. For example, for the following program:

```
var i: int;
var sum: int;
var a: int[10]; // adr(a) = FP-48

fn main() {
    i = 0;
    while (i < 10) {
        read a[i];
        i = i + 1;
    }

    i = 1;
    sum = 0;
    while (i < 10) {
        sum = sum + (a[i] - a [i - 1]);
        i = i + 1;
    }
    write sum;
}
```

The output could be roughly as follows (note, this is just for orientation):

```
1 ------------- left:3 -- right:2 -- dom:0 -- pred:
3 ------------- left:4 -- right:0 -- dom:1 -- pred: 1
    4: i = 0
4 --- while --- left:5 -- right:6 -- dom:3 -- pred: 3 5
    6: i PHI [i4 i19]
   16: ST ,   a
    8: i6 CMP 10
    9: (8) BGE 6
5 ------------- left:0 -- right:4 -- dom:4 -- pred: 4
   11:  RD
   12: i6 * 4
   13: FP + -48
   14: (13) + (12)
   15: ST (14), (11)  a
   18: i6 + 1
   19: i = (18)
   20: BR 4
6 ------------- left:7 -- right:0 -- dom:4 -- pred: 4
   22: i = 1
```

```
   23: sum = 0
7 --- while --- left:8 -- right:9 -- dom:6 -- pred: 6 8
   25: i PHI [i22 i41]
   26: sum PHI [sum23 sum39]
   27: i25 CMP 10
   28: (27) BGE 9
8 ------------ left:0 -- right:7 -- dom:7 -- pred: 7
   30: i25 * 4
   31: FP + -48
   32: LD (31), (30)  a
   33: i25 - 1
   34: (33) * 4
   35: FP + -48
   36: LD (35), (34)  a
   37: (32) - (36)
   38: sum26 + (37)
   39: sum = (38)
   40: i25 + 1
   41: i = (40)
   42: BR 7
9 ------------ left:2 -- right:0 -- dom:7 -- pred: 7
   44: WRT sum26
2 ------------ left:0 -- right:0 -- dom:1 -- pred: 9 1
   45: i PHI [i25 ?]
   46: sum PHI [sum26 ?]
   17: ST ,   a
   47: RET
```

## 6. Optimizations

Using the IR of the previous step, implement *copy propagation* and *global common subexpression elimination*. This should result in roughly the following CFG:

```
1 ------------ left:3 -- right:2 -- dom:0 -- pred:
3 ------------ left:4 -- right:0 -- dom:1 -- pred: 1
4 --- while --- left:5 -- right:6 -- dom:3 -- pred: 3 5
    6: i PHI [0 (18)]
    8: i6 CMP 10
    9: (8) BGE 6
5 ------------ left:0 -- right:4 -- dom:4 -- pred: 4
   11:  RD
   12: i6 * 4
   13: FP + -48
   14: (13) + (12)
   15: ST (14), (11)  a
   18: i6 + 1
   20: BR 4
6 ------------ left:7 -- right:0 -- dom:4 -- pred: 4
7 --- while --- left:8 -- right:9 -- dom:6 -- pred: 6 8
   25: i PHI [1 (40)]
   26: sum PHI [0 (38)]
   27: i25 CMP 10
   28: (27) BGE 9
8 ------------ left:0 -- right:7 -- dom:7 -- pred: 7
```

```
   30: i25 * 4
   31: FP + -48
   32: LD (31), (30)  a
   33: i25 - 1
   34: (33) * 4
   36: LD (31), (34)  a
   37: (32) - (36)
   38: sum26 + (37)
   40: i25 + 1
   42: BR 7
9 ------------- left:2 -- right:0 -- dom:7 -- pred: 7
   44: WRT sum26
2 ------------- left:0 -- right:0 -- dom:1 -- pred: 9 1
   45: i PHI [i25 ?]
   46: sum PHI [sum26 ?]
   47: RET
```

## 6. Register Allocation

This step is optional and is meant to deepen you understanding if you have enough time.

Use the algorithm shown in the lecture to build an interference graph. While traversing the CFG you should also eliminate unused instructions. Color the graph assuming that you have 30 registers available (2 registers are reserved as scratch registers).

After graph coloring coalesce the operands of phi instructions so that the phi instructions become unnecessary. Then remove the phi instructions, possibly inserting register moves (i.e., lr or lc instructions) at the end of the predecessor blocks.

Finally print out the CFG using the assigned registers for the operands of the instructions. The result should roughly look as follows:

| *Output* | *Corresponds to* |
|---|---|
| `1 -------------` | |
| `3 -------------` | |
| `R3: LC 0` | `R3 = 0` |
| `4 -------------` | |
| `R0: R3 CMP 10` | `4: R0 = R3 CMP 10` |
| `: R0 BGE 6` | `BGE R0, 6` |
| `5 -------------` | |
| `R2: RD` | `R2 = RD` |
| `R0: R3 * 4` | `R0 = R3 * 4` |
| `R1: FP + -48` | `R1 = FP - 48` |
| `R0: R1 + R0` | `R0 = R1 + R0` |
| `: ST R0, R2` | `mem[R0] = R2` |
| `R3: R3 + 1` | `R3 = R3 + 1` |
| `: BR 4` | `BR 4` |
| `6 -------------` | |
| `R3: LC 1` | `6: R3 = 1` |
| `R4: LC 0` | `R4 = 0` |
| `7 -------------` | |

```
   R0: R3 CMP 10            7: R0 = R3 CMP 10
       : R0 BGE 9              BGE R0, 9
8 ------------
   R0: R3 * 4                  R0 = R3 * 4
   R1: FP + -48                R1 = FP - 48
   R2: LD R1, R0               R2 = mem[R1 + R0]
   R0: R3 - 1                  R0 = R3 - 1
   R0: R0 * 4                  R0 = R0 * 4
   R0: LD R1, R0               R0 = mem[R1 + R0]
   R0: R2 - R0                 R0 = R2 - R0
   R4: R4 + R0                 R4 = R4 + R0
   R3: R3 + 1                  R3 = R3 + 1
       : BR 7                  BR 7
9 ------------
       : WRT R4             9: WRT R4
2 ------------
       : RET                   RET
```

## 7. Testing

It is highly recommended to implement basic tests to check on the key functionality of the symbol table, conversion to IR, and the optimizations.

In compilers, it often is tedious to build up the necessary state for testing individual functions or algorithms. Therefore, it is common practice to rely on end-to-end tests. Here, this means that for given input programs, one would test that it creates the expected textual representation.

For optimizations, it's recommended to construct small test programs that show the optimization working in different scenarios.

The provided GitHub repository has a basic GitHub Action Setup, which can be used as a starting point to have automatic tests, but does not yet contain such end-to-end tests.

## 7. Basic Marking Rubric

As a very rough guide to the marking, here a basic rubric. This is also an indication of what we will roughly ask during the project presentations. To raise the exam mark by 1, only the elements categorized as good are strictly required.

| | good | very good |
|---|---|---|
| Symbol Table and Language Correctness | Support all correct programs. | Give useful errors for incorrect programs. |
| IR Generation and Optimizations | Generate a correct CFG and SSA form for the given programs. A notable attempt at implementing optimizations. | Implements the optimizations for the given programs. |
| Register Allocation | | A good attempt at the inference algorithm and graph coloring. |

## Plagiarism and Duplication of Material

The work you submit must be your own. We will run checks on all submitted work to identify possible plagiarism or other academic misconduct and take disciplinary action against anyone found to have broken the University's rules on academic integrity.